ACCT: An Intelligent Congestion Control Mechanism for Future Internet

Morteza Kheirkhah*, Joao Reis†, Fan Yang†, David Griffin† and Miguel Rio†
Samsung Research UK*, University College London†

***Abstract*—We propose ACCT, an end-to-end learning-based congestion control mechanism for Internet flows that aims to achieve an optimal sending rate through collaboration between applications and the network elements along the network path. Specifically, an ACCT sender can exploit explicit feedback from the network about its ability to satisfy the application's desired sending rate. Unlike existing schemes, ACCT does not blindly adjust its sending rate to either of these signals; instead, it considers them advisory because the bottleneck may be at routers that do not support ACCT signaling. To detect such bottlenecks and strictly control the end-to-end delay for latency-sensitive applications, ACCT actively measures the end-to-end queuing delay and regulates its sending rate to keep queuing delay within an acceptable margin. Finally, ACCT heavily relies on a Reinforcement Learning (RL) agent to adjust the aggressiveness of its sending rate. We propose a novel approach to formulate the RL agent, exploiting multiple reward functions simultaneously to train the agent. We implemented the ACCT congestion control module in NS-3, which communicates with the RL agent via a technology-agnostic protocol. We evaluated performance across various network scenarios in both wired and wireless networks. Across all experiments, ACCT significantly outperforms commonly used TCP variants such as CUBIC; it provides network fairness when competing with other ACCT flows, and finally, ACCT flows coexist well when they carry heterogeneous traffic.**

I. Introduction

Traditional end-to-end rate control approaches such as those mechanisms in TCP and its variants [1], [2], [3] cannot meet the requirements of emerging Internet applications that typically require high-bandwidth and low-latency simultaneously. The key reason behind this shortcoming is that end-host-centric approaches are primarily designed to achieve high network throughput with less regard for achieving end-to-end latency targets. As a result, applications with latency constraints rely on the network to meet their requirements by means of traffic engineering. This problem is aggravated over wireless networks due to wireless being intrinsically a shared medium and also because wireless channels are less stable than wired networks, fluctuating rapidly in the order of milliseconds [4], [5], [6], [7], [8], [9]. The degree of fluctuation is worse with emerging high capacity wireless technologies that operate at high-frequency bands such as mmWave [10], [11]. This can result in rapid changes in the available capacity to wireless users. Traditional end-host-centric approaches, which are heavily reliant on end-to-end measurements to regulate their sending rate, fail to track these rapidly varying capacity changes. Consequently either the radio resources are under-utilized or excessive queues are built up in the last mile hop. Even BBR [12], the state-of-art end-host-centric, rate-based, congestion control scheme, fails to precisely track such capacity changes [13]. Although BBR can detect when the pipe is not fully utilized, it typically fails to estimate the available capacity precisely. As a result, it blindly increases its sending rate to saturate an underutilized pipe. This can result in long queuing delays [13], [14]. To tackle these shortcomings, we argue that a precise rate control across a diverse set of future Internet flows, carrying traffic of multiple applications with differing requirements, can only be achieved when end-hosts (applications) and networks collaborate. This way, applications and networks can express their desired requirements to one another, allowing applications to adapt themselves to network constraints and the networks to prioritize their resource distribution according to the applications' requirements.

There are a few existing network-centric proposals that try to adopt a collaborative approach between senders and the network, such as XCP [15] and RCP [16]. Each sender expresses its desired sending rate and other feedback information, such as the RTT measurement to routers. Each router along the network path provides multiple bits of feedback per packet to senders, indicating the sending rate that senders should follow. This way, senders do not need to be concerned about adjusting their sending rates and preserving network fairness with other competing flows as the network deals with these aspects. However, delegating these responsibilities to the network introduces two problems: (1) the approaches do not scale well with high capacity networks with large numbers of flows due to complex operations that need to be performed on a per-packet basis; and (2) all network elements in all networks along the path should support the proposed signaling mechanisms. These problems have hindered the deployment of these proposals over the past two decades. Recent proposals such as ABC [13] and PBE-CC [14] have returned to these schemes but with an approach of making them more distributed. These schemes, however, are particularly designed for wireless networks, and thus the focus of their design is only at the wireless link and the last mile hop bottleneck (*i.e.,* base stations and/or Wi-Fi APs). Both ABC and PBE-CC fall back to CUBIC [1] and BBR, respectively, if they detect non-wireless bottlenecks. Additionally, these proposals do not consider the application requirements at all, and thus they rely on the network to engineer traffic of applications with different requirements.

*Part of this article was written by Dr. Morteza Kheirkhah while he was associated with University College London and not Samsung's employee.

We propose ACCT, an end-to-end learning-based rate control mechanism for Internet flows which aims to achieve an optimal sending rate by collaborating with network elements on the network path (*e.g.,* routers) and the application simultaneously. Specifically, an ACCT sender can exploit explicit feedback from the network about the available capacity and the application about the desired sending rate. Unlike existing schemes, ACCT does not blindly adjust its sending rate to either of these signals; instead, it considers them advisory because the bottleneck may be at routers that do not support ACCT signaling. To detect such bottlenecks and strictly control the end-to-end delay for latency-sensitive applications, ACCT actively measures the end-to-end queuing delay and regulates its sending rate within a small queuing delay margin. Finally, ACCT heavily relies on a Reinforcement Learning (RL) agent to adjust the aggressiveness of its sending rate (unlike existing explicit rate control schemes such as XRC [17] and WhiteHaul [18]). We propose a novel approach to formulate the RL agent, exploiting multiple reward functions simultaneously to train the agent. Each reward function defines a strategy for ACCT to follow within an operating zone or a defined condition (see § II-B). Recent learning-based proposals such as Aurora [19] and Ocra [20] only follow a single reward function and adjust the congestion window size rather than the flow aggressiveness, which mainly limits them to be applicable to different networks [19]. Our key contributions are as follows:

- We design a new end-to-end congestion control algorithm that utilizes feedback from the network to optimize the sending rate (§ II-A). ACCT defines four different operating zones, which are detected based on explicit feedback from the network and also the state of E2E queuing delay (§ II-B). ACCT adjusts the aggressiveness of its sending rate predictively with the help of an RL agent.
- We formulate the RL agent in which it uses a separate reward function for each of operating zones (§ II-C). This approach teaches the RL agent to follow a different strategy in each operating zone . As a result, it allows a well-trained neural network (NN) model to be used under a wide range of conditions.
- We implement ACCT in NS-3. The ACCT congestion control module is written in C/C++, while the RL agent is in Python. We also modified the NS-3 networking stack of wired and LTE networks to enable ACCT signaling.
- Our prototype is used to evaluate the performance of ACCT across various scenarios and network settings in both wired and wireless network topologies (§ III).

II. ACCT Design

ACCT is designed to meet the following objectives:

- To distribute the available resources at the network routers with awareness of application requirements such as the desired sending rate.
- To track available capacity along the network path by (1) consuming the advisory feedback from ACCT-enabled routers; (2) observing end-to-end network performance to detect bottlenecks that are at non-ACCT enabled routers.
- To maintain end-to-end queuing delay within a small acceptable margin to meet the stringent latency requirements of emerging applications.
- To coexist with other commonly used TCP variants (such as CUBIC) in case ACCT flows need to compete with them in a shared bottleneck.
- To adapt to a wide range of networks with different bandwidth-delay products (BDP) automatically without requiring explicit parameter adjustment for different BDP networks within the transport layer (*e.g.,* tuning the flow aggressiveness for increase congestion control function).

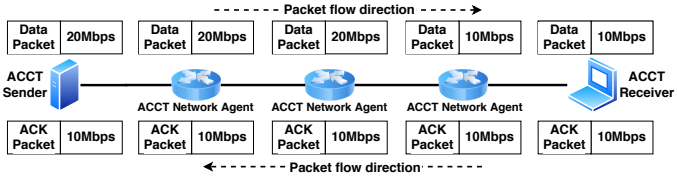
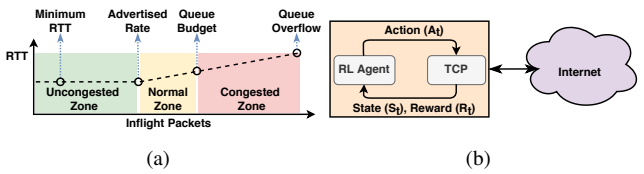


Figure 1. An overview of the ACCT deployment scenario. The sender expresses its desired sending rate. The Network Agent extracts this information from the data packet and overwrites it if a smaller rate can be offered. The ACCT receiver piggybacks this feedback on ACK packet. The ACCT sender considers this feedback and regulates its sending rate by help of an RL agent.

ACCT has three main components. (i) The **ACCT Network Agent** is a distributed subsystem that resides at the network routers. (ii) The **ACCT Sender** is an end-to-end congestion control algorithm that operates at the transport layer. (iii) The **ACCT RL Agent** is responsible for adjusting the aggressiveness of ACCT flows at the sender's side.

A. ACCT Network Agent

At connection startup or whenever an application's requirements change, the ACCT sender expresses the application's desired sending rate in its data packets (*e.g.,* through an IP or TCP option). When an ACCT Network Agent (NA) at a router receives a data packet with that signal, it first checks whether it can provide the requested rate. If not, the NA overwrites the information in the data packet with a new smaller rate. This process continues router-by-router until the data packet reaches the receiver. The receiver then piggybacks this feedback in the ACK packet returned to the sender. Fig. 1 illustrates these interactions. The NA at the 3rd router overwrites the received data packet with a new smaller rate (10Mbps), indicating that sending more than 10Mbps by this flow may cause the queue to build up at this router. Unlike existing schemes such as XCP and RCP, the NA only calculates how its capacity should be distributed among competing flows without directly controlling queue occupancy. Instead, the ACCT sender is responsible for adjusting its sending rate in response to the signal of available capacity received from the network and also the locally measured end-to-end queuing delay. This way, the sender is not required to fully comply with the sending rate proposed by the network and also a network router does not need to accurately calculate a sending rate for the sender. This loosely-coupled interaction between these components simplifies the rate calculation at routers and also permits the



(a) (b)

Figure 2. (a) ACCT operating zones. (b) ACCT high-level architecture.

end-to-end rate control mechanism to be performed by end-hosts in a distributed manner. An ACCT NA follows a simple capacity estimation in which it splits the link capacity among the set of active flows traversing it. Also, the NA considers a flow as inactive when it does not receive a data packet from it for a certain time (*e.g.,* 250ms, more than the TCP timeout). The NA marks a flow active when gets a data packet from it.

B. ACCT Sender

Several TCP variants (e.g., CUBIC, BBR) try to tackle the problem of slow increase of TCP by adopting a Multiplicative-Increase Multiplicative-Decrease (MIMD) policy. However, these variants increase their sending rate blindly, causing bottleneck queues to build up or overflow, resulting in an increased end-to-end delay [13]. ACCT is designed to prevent such delays by an increase function that is tuned intelligently by an RL agent (§ II-C). The following describes how ACCT modifies TCP window increase and decrease functions:

- For a non-duplicated ACK, increase the window (w) by $\frac{\alpha}{w}$
- For a loss packet, decrease the window (w) by $\frac{w}{\beta}$

The value of α is 1 with TCP, which increases the congestion window (w) by one segment every window of data (roughly an RTT). If the value of α is greater than 1 then w is increased more aggressively by more than one segment per RTT, while if it is smaller than 1, w is increased more slowly, less than one segment per RTT. With ACCT, α is dynamically adjusted by an RL agent. The value of β is 2 with TCP and ACCT.

A key question here is how ACCT utilizes the explicit feedback it receives from the network? First, ACCT splits its operating zone into three distinct zones based on the explicit feedback it receives from the network and its periodic end-to-end measurements of queuing delay. Fig. 2(a) illustrates these operating zones and the corresponding measurement indications that detect them. Flows are in the Green Zone when ACCT operates on a path formed from unsaturated links where adding new segments does not increase the RTT. ACCT detects this zone when its current sending rate is below that advertised by the network and also when its estimated average queuing delay (which is a function of RTT) is smaller than a certain threshold. The Yellow Zone begins when the bottleneck buffer starts to build up queue, and it lasts until the queuing budget has been exceeded. ACCT detects this zone when its queuing delay estimation is smaller than the queuing budget and also its sending rate is above the advertised rate from the network. The Red Zone starts when the estimated queuing delay at the sender is above the queuing budget. The Red Zone continues until the bottleneck buffer overflows and packets start being dropped (indicated by the circle in the top right of Fig. 2(a)).

The key intuition behind this zone separation is that ACCT needs to adopt different behaviors/strategies in each zone. For example, when the Green Zone is detected, ACCT should not be concerned about the queuing delay and it can aggressively and predictively increase its sending rate to prevent link under-utilization while not increasing the end-to-end delay. On the other hand, when ACCT detects the Yellow (or Red) Zone it should not worry about link under-utilization, but instead, it should follow a gentle increase in order to continuously probe the network and monitor the impact on the end-to-end delay. On that basis, the ACCT Sender regularly informs the RL Agent of the current operating zone. The RL Agent follows different strategies by applying different reward functions when in each operating zone (§ II-C). Additionally, the definition of the operating zones assists ACCT with the detection of non-ACCT bottlenecks. ACCT detects such a bottleneck when the queuing delay estimation is higher than the queuing budget while the sending rate is lower than the advertised rate from the network. To signal this detection to the RL agent, ACCT also defines another operating zone so-called Blue Zone (see § II-C for more details).

End-to-end measurements. ACCT continuously tracks three parameters: minimum RTT (RTT_{min}), target congestion window ($twnd$), and average queuing delay (σ). The latter is updated once every window of data (*i.e.,* roughly an RTT), while the first two parameters are updated upon the arrival of every ACK packet. Whenever an ACK is received with explicit rate information from the network, the ACCT Sender calculates the target sending window ($twnd$) as follows:

$$twnd = R_{net} * RTT_{min} \tag{1}$$

Where R_{net} is the advertised rate by the ACCT Network Agents and RTT_{min} is the minimum observed RTT since connection startup. In an ideal condition, $twnd$ should maximize throughput with no queuing delay. However, adjusting $cwnd$ blindly to $twnd$ is not practical as the bottleneck can be at a router that does not support ACCT's explicit feedback mechanism. Thus, as discussed earlier, monitoring queuing delay is crucial for detecting such a bottleneck and to avoid simply following $twnd$, which has not taken the real bottleneck into account. ACCT estimates the queuing delay (σ), every window of data, with the following formula:

$$\sigma = \sum_{j=1}^{N} RTT_j * \frac{1}{N} - RTT_{min} \tag{2}$$

Where σ is the queuing delay estimation, RTT_j is jth RTT sample and N is the total number of RTT samples in a window.

Periodic window reduction. ACCT relies on an RL algorithm to calculate a weight factor (α), which dictates the aggressiveness of the ACCT sender in each zone. While this is crucial to prevent link under-utilization across various networks with different BDP, ACCT also employs a periodic window reduction to strictly control end-to-end latency. The minimum value of α that is decided by the RL agent is not less than one packet, so ACCT increases its $cwnd$ at least

by one segment every RTT to ensure continuous probing of the network. As a result, ACCT will enter the Red Zone eventually. To keep ACCT within the Yellow Zone, ACCT resets the $cwnd$ to the estimated target window ($twnd$) when the end-to-end queuing delay exceeds the pre-defined queuing budget. The strategy of the RL Agent during the Yellow Zone is to reach this reduction point slowly (see II-C).

C. ACCT RL Agent

RL Architecture. Fig. 2(b) illustrates the architecture of the RL agent. The agent interacts with the TCP module, receiving a set of state information (S_t) as well as a reward value (R_t) in each regular time interval (e.g. every window of data). The agent inputs this information into a NN which results in an action (A_t) (*i.e.,* a value for α). The communication between the TCP module and RL Agent can be achieved through the Netlink socket. Finally, the RL agent uses the A3C framework [21] which has shown a promising approach for solving networking problems in several prior works [22].

RL Formulation. An RL technique requires the design of three aspects: state inputs, reward functions and action space.

State inputs. The TCP module sends the following inputs to the RL agent to express various network conditions:

$$S_i = \{[T_i],\ L_i,\ [\frac{D_i}{DB}],\ A_{i-1},\ [Zone_i],\ [\frac{cwnd_i}{twnd_i}]\} \quad (3)$$

Where T_i is the throughput of window i (MB/s) that is calculated by dividing the ACKed bytes in a window by the mean of the RTT samples in that window. L_i is the loss rate of window i (pkt/s). $\frac{D_i}{DB}$ is the end-to-end queuing delay, which is calculated by dividing the mean queuing delay observed in window i (D_i) by the queuing budget (DB). A_{i-1} is the action taken by the RL agent in the previous window. $zone_i$ indicates the operating zone of window i. Note that we hold a time series history (last 8 samples) for elements shown with square brackets. We use 1-D Convolutional Neural Networks (CNNs) for the time series components to convert the vectors of time series entries into single scalar values. Each state input is fully connected to a hidden layer with 128 neurons, each using a Relu activation function. Fig. 3 shows a representation of the RL agent's A3C neural networks.

Action space. the action space (as) is the set of possible values for α. In every window of data the RL agent chooses an action from this set (see below). Larger action spaces require longer training times, but we found 15 produces good results.

$$A_i = [1,\ 2,\ 3,\ 4,\ 5,\ 10,\ 20,\ 30,\ ...,\ 80,\ 90,\ 100] \quad (4)$$

Reward functions. The TCP module sends a reward to the RL agent to indicate how good was the last action taken by the agent. During the course of training, the RL agent aims to maximize the accumulated reward by taking action in a particular state (network condition) which results in a higher value of reward in future states. In other words, the RL agent tends to adjust the flow aggressiveness (α) so that TCP stays in states resulting in a high reward. Unlike existing schemes [19], [20], ACCT does not follow a single reward function to train its NN model. Instead, it defines four different operating zones and utilizes a different reward function within each zone (see § II-B). In the Green Zone, ACCT gets high reward if it increases its sending rate largely with the following function:

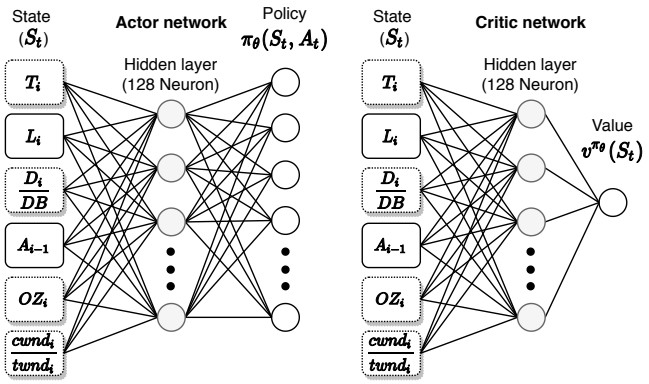


Figure 3. A3C [21] has two neural networks, and both networks receive the same set of state inputs (S_t). The output of the Critic network is a value function which refines the Actor network policy ($v^{\pi_\theta}(S_t)$). The output layer of the Actor network produces a probability value for each action. Boxes with dotted borders are 1-D Convolutional Neural Networks (CNNs).

$$R_G = A_{i-1} - \omega L_i \quad (5)$$

Where A_{i-1} is the action taken by the agent in the last window. L_i is the number of lost packets in the current window. The ω was selected experimentally to be 10. We formulate other reward functions to reach a similar maximum value when the agent behaves optimally in the other operating zones. The Yellow Zone is the optimal operating zone. The reward function is designed to prefer conservative increases in sending rate, which tends to keep operations within the Yellow Zone for longer, increasing accumulated reward:

$$R_Y = (as[N-1] + 1 - A_{i-1}) - \omega L_i - \zeta B_i \quad (6)$$

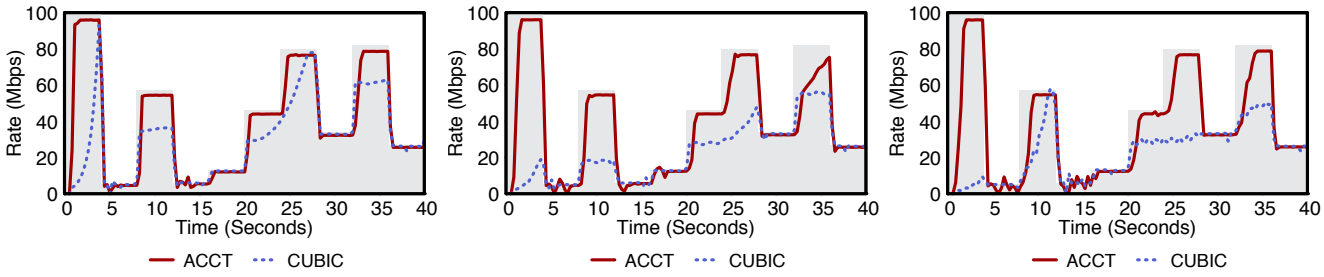
Where N is the total number of available actions in the action space (as). B_i indicates how bursty the sending rate is in the current window compared to the previous one. To prevent a large negative value for this parameter, we calculate it via a scaled-down as, with values ranging from 1 to 2.4. The value of ω and ζ are 10 and 1, respectively. This reward function pushes the agent to be more conservative during this zone by producing a higher reward when the agent reduces the alpha value ($\alpha \rightarrow 1$). In the Red Zone, ACCT follows a similar reward function to the one in the Yellow Zone; Finally, when ACCT detects a non-ACCT bottleneck (see § II-B), it enters the Blue Zone and pursues the following:

$$R_B = (as[N-1] + 1 - A_{i-1}) - \beta\frac{D_i}{DB} - \omega L_i - \zeta B_i \quad (7)$$

Where $\frac{D_i}{DB}$ is the measured end-to-end queuing delay.

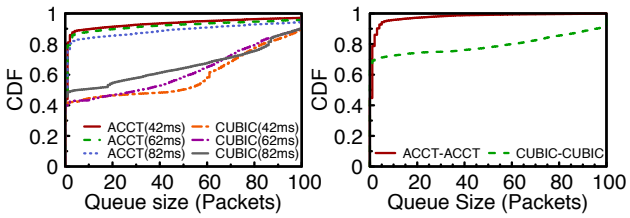
III. Evaluation

Experiment setup. We conducted our experiments over both wired networks and wireless networks with various link capacities and RTTs. With the wired network (Fig. 6(a)), links



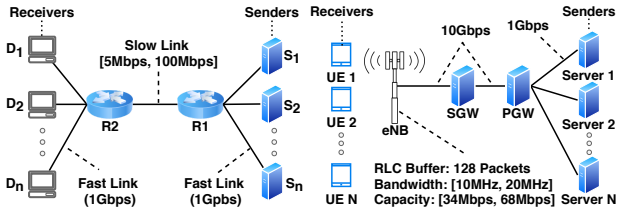
(a) RTT: 42ms, BDP: 525KB (b) RTT: 62ms, BDP: 775KB (c) RTT: 82ms, BDP: 1MB

Figure 4. Comparing ACCT with CUBIC across several highly dynamic network conditions over a wired topology. The gray shaded area shows the bottleneck's link capacity, which changes every 4 secs.



(a) (b)

Figure 5. (a) The queue size distribution (CDF) for experiments in Fig. 4. At 80th percentile ACCT only occupies the queue by less than 5 pack whereas CUBIC by more than 80 pkts. (b) Red and green curves show the queue size CDF of experiments in Fig. 9(a) and Fig. 9(b) respectively. ACCT controls the queuing delay within its queuing budget of 5ms at 99% of the time.



(a) (b)

Figure 6. (a) Wired network topology. (b) LTE cellular network topology.

between hosts and routers had a capacity of 1Gbps. We changed the one-way delay of these links to model different E2E propagation delays. The link between the routers had 1ms delay, but we varied its capacity across experiments to model network congestion. With the wireless network (Fig. 6(b)), we used a single base station with a total bandwidth of $20MHz$.The RLC buffer size was configured with 128 packets. In all experiments, we enabled TCP SACK and Timestamp options, the queue budget of ACCT was 5ms. We turned off the TCP slow start mechanism at connection startup across all schemes (unless stated otherwise). Finally, the desired rate expressed by the application to the TCP socket was set to a large value unless stated otherwise.

Efficiency and responsiveness. This experiment studies how quickly ACCT can ramp up and down its sending rate when it detects changes in capacity across various networks with different BDPs. We used our wired network topology (Fig. 6(a)). Every 4 secs, we randomly changed the maximum capacity of the link between routers, ranging from 5Mbps to 100Mbps. Fig. 4 shows the results, comparing ACCT with CUBIC in different network configurations. When BDP is small, Fig. 4(a), CUBIC ramps up its sending rate faster but as BDP increases, Fig. 4(b) and 4(c), CUBIC requires more than 4 secs to fully saturate the available capacity due to its slow cubic function behaviour. This shortcoming of CUBIC can be seen in the first 4 secs of each experiment, where CUBIC needs to increase its sending rate blindly in the absence of the TCP slow start mechanism. On the other hand, ACCT increases its sending rate quickly at connection startup and when the available capacity suddenly increases (*i.e.,* when within the Green Zone). Thanks to the RL agent, which proactively regulates the ACCT aggressiveness. When the available capacity is suddenly reduced, ACCT follows the capacity feedback it receives from the bottleneck router and accurately adjusts its sending rate. Fig. 5(a) shows the queue size distribution across these experiments. For 80% of the time, ACCT occupies fewer than 5 packets, whereas this is 80 packets with CUBIC that reduce its sending rate only when it detects packet losses. *i.e.,* the bottleneck's buffer overflows before CUBIC can reduce its sending rate. If many packets get dropped due to a sudden reduction in the available capacity, then costly retransmission timeouts become inevitable.

Fairness and Convergence Multiple TCP flows sharing the same bottleneck link should be able to compete fairly with one another. We thus study how quickly ACCT converges to its fair share of capacity as flows join and leave the wireless bottleneck. For these experiments, we use the wireless network topology (Fig. 6(b)). Note that we only trained the ACCT RL agent with wired networks, meaning that the RL agent has never been trained with cellular networks. We consider F ACCT flows (*i.e.,* $F = 4$) sharing a cellular bottleneck. Initially, 4 ACCT flows arrive at the UE with an inter-arrival time of 10 secs, making the base station a bottleneck. After 40 secs, the flows leave the bottleneck again with 10 secs gap between each flow's departure. Fig. 7 shows the results for scenarios where the RTT of flows is identical (7(a) & 7(b)) and different (7(c) & 7(d)). ACCT flows share bottleneck capacity equally as flows join and leave regardless of their RTT; ACCT converges to its fair-share quickly while it controls the RLC's Head-of-Line (HoL) delay within the budget of 5ms at 99% of the time in both experiments. Fig. 7(b) and 7(d) show the RTT measurements at the senders. These figures show the RTT of each flow and how the RL agent carefully adjusts the ACCT

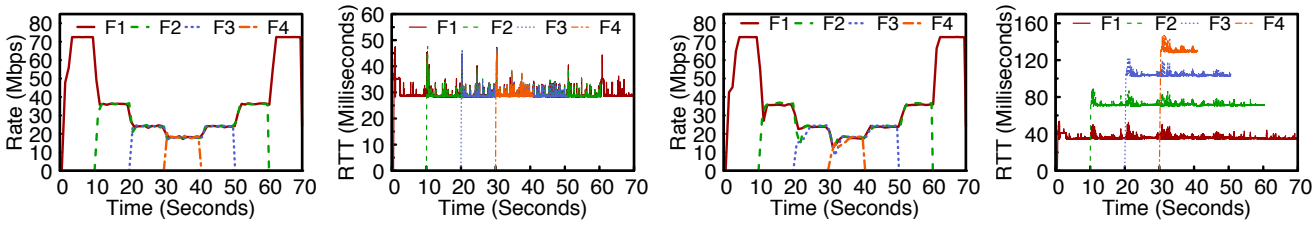


Figure 7. Fairness among competing ACCT flows when they have (a) similar RTTs and (c) different RTTs. Figures (b) and (d) plot the observed RTT as a time-series at the ACCT senders for different cases of RTT similarity among flows. Experiments are conducted on our wireless network topology (Fig. 6(b)).

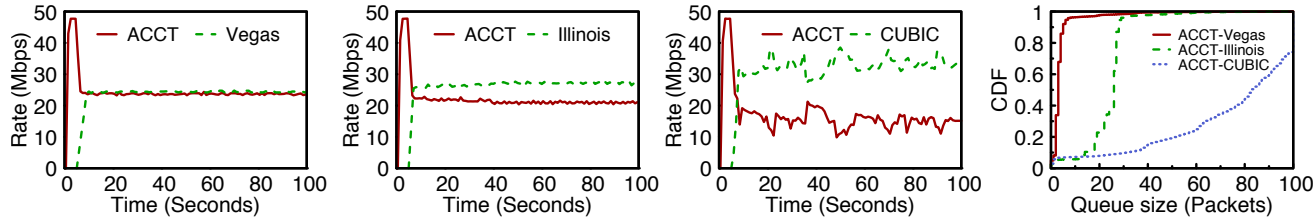


Figure 8. Achieved sending rate when (a) ACCT competes with Vegas, (b) ACCT competes with Illinois, (c) ACCT competes with CUBIC.

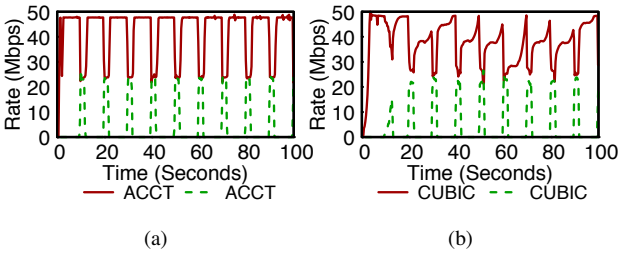


Figure 9. Comparing ACCT to CUBIC for the coexistence of heterogeneous traffic. The red curves represent file download traffic and the green curves represent video streaming traffic with an ON/OFF traffic pattern.

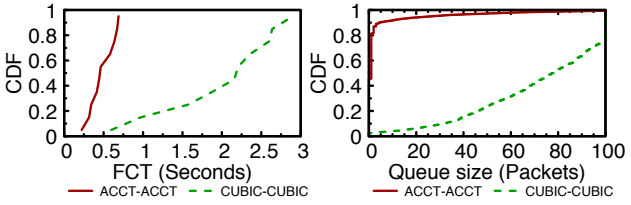
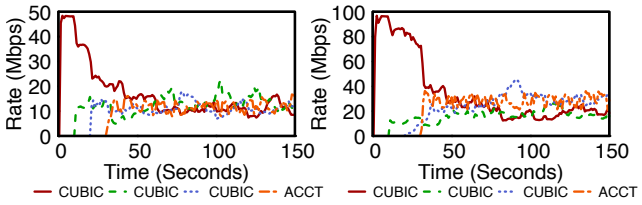


Figure 10. Comparing low-latency flows for the (a) FCT and (b) queue size.

aggressiveness to stay in the Yellow Zone for longer to achieve a higher accumulated reward.

Coexistence We consider how well ACCT flows coexist when they are carrying heterogeneous traffic. We first define our flow types and then walk through the experiments. *Bandwidth-hungry flow.* A long-lived TCP flow remains in the bottleneck link for the duration of the simulation. This flow represents file download traffic. *Video-streaming flows.* A TCP flow with an ON/OFF traffic pattern. The TCP flow has a periodic cycle of 10-secs where in the first 2-secs it bursts packets with constant bit rate of $23Mbps$ (*i.e.,* representing the transmission of a high quality video chunk), then stops sending for the next 8-secs if all the sent packets during the ON period are delivered successfully. This 10-secs cycle is repeated again for the period of the simulation. *Latency-sensitive flow.* A medium-size TCP flow with random sizes ranging from $500KB$ to $2MB$. During the simulation period, we generated 20 non-overlapping flows



(a) Bottleneck capacity: 50Mbps (b) Bottleneck capacity: 100Mbps

Figure 11. ACCT competes with CUBIC at a non-ACCT bottleneck.

with an inter-arrival rate of 5 secs.

Bandwidth-hungry vs. video-streaming flows. An ACCT flow carrying file download traffic competes with another ACCT flow carrying video traffic. We repeat the same experiment with CUBIC. Fig. 9 shows the results. ACCT shows immediate response and successfully slows down the rate of the file download traffic (red curve) as the video-streaming traffic (green curve) bursts its packets into the network. When the video traffic enters its OFF period, ACCT immediately increases its sending rate, fully utilizing the available capacity. On the other hand, CUBIC fails to slow down and consequently, the bursty video traffic causes severe packet losses across both competing flows. As a result, the video chunk may take longer than 2 secs to be delivered with non-ACCT flows, degrading users' QoE watching the video content. These losses can be clearly seen in Fig. 9(b) because CUBIC rapidly reduces its rate. This figure also highlights that file download flow is incapable of ramping up its sending rate quickly when the video flow enters its OFF period. Finally, Fig 5(b) shows that ACCT consistently (for 99% of the time) maintains the queuing delay below its budget of 5ms, almost 5 times lower than CUBIC.

Bandwidth-hungry vs. low-latency flows. A bandwidth-hungry ACCT flow competes with 20 non-overlapping low-latency ACCT flows, each joining the bottleneck with an inter-arrival gap of 5 secs. Fig. 10(a) shows the results for the flow completion time (FCT) of low-latency flows. With ACCT the

FCT of low-latency flows is reduced by 5x and 3x at the 50th percentile and 99.9th percentile, respectively, compared with CUBIC. This reduction in FCT can also be seen in Fig. 10(b) which shows the bottleneck queue size distribution for these experiments. At the 75th percentile, CUBIC overflows the queue of the network device and starts building queue at queue disc while ACCT occupies 1 packet.

TCP-friendliness. ACCT should coexist with other used TCP variants in the same bottleneck. To examine this, we used our wired network topology and compared the performance of ACCT when it competes with TCP Vegas (delay-based), Illinois (delay-based), and CUBIC (loss-based) at a shared ACCT-enabled bottleneck. Fig. 8 shows the results. When ACCT competes with delay-based TCP variants, the network fairness between flows can be preserved while the queuing delay is gracefully controlled. Note that the queuing delay can't be controlled strictly in such scenarios because the behavior of other traffic cannot be fully controlled by ACCT;*e.g.,* Illinois tends to occupy more queues than Vegas (Fig. 8(d)). When ACCT competes with CUBIC, it achieves its fair share of resources, but CUBIC compromises the end-to-end delay.

Non-ACCT bottleneck. We explore scenarios where the bottleneck is at a non-ACCT enabled router. We used a wired topology and turned off the ACCT signaling capability at routers R1 and R2. A new flow is established every 10 secs: the first three flows are CUBIC, and the last flow is ACCT. Fig. 11 shows the results for two network scenarios with different capacities of the bottleneck link. The ACCT flow in Fig. 11(a) and 11(b) has a desired rate of 20Mbps and 40Mbps, respectively, which is significantly greater than its fair-share of the bottleneck (*i.e.,* 12.5Mbps and 25Mbps, respectively). In both experiments, ACCT joined the bottleneck gracefully and shared the resource well with CUBIC flows. Note that the CUBIC flows suffered from slow convergence. This is visible in the first 30 secs. When ACCT joined the bottleneck, it caused all flows to converge faster. The reason is that the RL agent causes small bursts by temporarily increasing α when CUBIC flows significantly occupy the bottleneck's buffer; *i.e.,* the RL agent tries to stay in the Green Zone, collecting more rewards rather than getting stuck in the Blue Zone. The only way to reduce the sending rate of CUBIC flows under these conditions is when CUBIC observes packet losses.

IV. Conclusions and Future Plan

We present ACCT, an end-to-end, learning-based, congestion control algorithm for Internet flows. ACCT senders express their desired sending rates to routers and ACCT-enabled routers provide explicit feedback to the senders about their maximum sending rates. Unlike XCP, ACCT considers the network feedback advisory and follows it cautiously by also actively monitoring the end-to-end network conditions in case the bottleneck is at routers that do not support ACCT.

We evaluated ACCT performance under various scenarios in wired and wireless networks in NS-3. ACCT significantly outperforms CUBIC; ACCT provides network fairness when competing with either ACCT flows or non-ACCT flows at both ACCT-enabled and non-ACCT routers; and, finally, ACCT flows coexist well when they carry heterogeneous traffic.

Finally, we are currently working on a real prototype of ACCT with Linux Kernel supporting several state-of-the-art congestion control algorithms. Additionally, we plan to design a multipath variant of the ACCT with existing multipath transport protocols such as MPTCP [23] and its variants [24], [25], [26], [27]. We will report on these in the near future.

References

[1] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS operating systems review*, 2008.

[2] L. S. Brakmo, S. W. O'malley, and L. L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," in *SIGCOMM '94*.

[3] S. Liu, T. Başar, and R. Srikant, "TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks," *Performance Evaluation*, vol. 65, no. 6-7, pp. 417–440, 2008.

[4] H. Lee, J. Flinn, and B. Tonshal, "RAVEN: Improving Interactive Latency for the Connected Car," in *MobiCom '18*.

[5] W. K. Leong and et al., "TCP Congestion Control Beyond Bandwidth-Delay Product for Mobile Cellular Networks," in *CoNEXT '17*.

[6] S. Park, J. Lee, and et al., "Exll: an extremely low-latency congestion control for mobile cellular networks," in *CoNEXT '18*.

[7] F. Lu and H. e. a. Du, "CQIC: Revisiting Cross-Layer Congestion Control for Cellular Networks," in *HotMobile '15*.

[8] Y. Zaki, T. Pötsch, and et al., "Adaptive Congestion Control for Unpredictable Cellular Networks," in *SIGCOMM '15*.

[9] K. Winstein and et al., "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks," in *NSDI '13*.

[10] M. Zhang and et al., "Will TCP work in mmWave 5G cellular networks?" *Communications Magazine*, vol. 57, no. 1, pp. 65–71, 2019.

[11] M. Polese, R. Jana, and M. Zorzi, "TCP in 5G mmWave networks: Link level retransmissions and MP-TCP," in *INFOCOM WKSHPS*, 2017.

[12] N. Cardwell and Y. e. a. Cheng, "BBR: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.

[13] P. Goyal, A. Agarwal, and et al., "ABC: A Simple Explicit Congestion Controller for Wireless Networks," in *NSDI '20*.

[14] Y. Xie and et al., "PBE-CC: Congestion Control via Endpoint-Centric, Physical-Layer Bandwidth Measurements," in *SIGCOMM '15*.

[15] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *SIGCOMM '02*.

[16] N. ukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the internet," in *IWQoS '05*.

[17] M. Kheirkhah, M. M. Kassem, G. Fairhurst, and M. K. Marina, "XRC: An Explicit Rate Control for Future Cellular Networks," in *ICC 2022 - IEEE International Conference on Communications*, 2022.

[18] M. M. Kassem, M. Kheirkhah, M. K. Marina, and P. Buneman, "WhiteHaul: An Efficient Spectrum Aggregation System for Low-Cost and High Capacity Backhaul over White Spaces," in *MobiSys '20*.

[19] N. Jay, N. Rotman, B. Godfrey, and et al., "A deep reinforcement learning perspective on internet congestion control," in *ICML '19*.

[20] S. Abbasloo and et al., "Classic meets modern: a pragmatic learning-based congestion control for the internet," in *SIGCOMM '20*.

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, and et al., "Asynchronous methods for deep reinforcement learning," in *ICML '16*.

[22] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *SIGCOMM '17*.

[23] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *ASIGCOMM '11'*.

[24] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and Evaluation," in *CoNEXT '17*.

[25] M. Kheirkhah and M. Lee, "AMP: An Adaptive Multipath TCP for Data Center Networks," in *2019 IFIP Networking Conference (IFIP Networking)*.

[26] M. Kheirkhah, I. Wakeman, and G. Parisis, "MMPTCP: A multipath transport protocol for data centers," in *INFOCOM '16*.

[27] Y. Cao, M. Xu, X. Fu, and E. Dong, "Explicit Multipath Congestion Control for Data Center Networks," in *CoNEXT '13*.